~~DTIC FILE COPY~~

(2)

MASSACHUSETTS INSTITUTE OF TECHNOLOGY VLSI PUBLICATIONS

VLSI Memo No. 89-531
May 1989

Work-Preserving Emulations of Fixed-Connection Networks

Richard Koch, Tom Leighton, Bruce Maggs, Satish Rao, and Arnold Rosenberg

Abstract

In this paper, we study the problem of emulating T_G steps of an N_G-node guest network on an N_H-node host network. ~~We~~ call an emulation *work-preserving* if the time required by the host, T_H, is $O(T_G N_G / N_H)$ because then both ~~the~~ guest and host networks perform the same total work, $\Theta(T_G N_G)$, to within a constant factor. We say that an emulation is *real-time* if $T_H = O(T_G)$, because then the host emulates the guest with constant delay. Although many isolated emulation results have been proved for specific networks in the past, and measures such as dilation and congestion were known to be important, the field has lacked a model within which general results and meaningful lower bounds can be proved. We attempt to provide such a model, along with corresponding general techniques and specific results in this paper. Some of the more interesting and diverse consequences of this work include:

(KR)

1. a proof that a linear array can emulate a (much larger) butterfly in a work-preserving fashion, but that a butterfly cannot emulate an expander (of any size) in a work-preserving fashion,
2. a proof that a mesh can be emulated in real time in a work-preserving fashion on a butterfly, even though any $O(1)$-to-1 embedding of a mesh in a butterfly has dilation $\Omega(\log N)$,
3. a proof that an $N \log N$-node butterfly can be emulated in a work-preserving fashion on an N-node shuffle-exchange graph, and vice-versa,
4. simple $O(N^2/\log^2 N)$-area and $O(N^{3/2}/\log^{3/2} N)$-volume layouts for the N-node shuffle-exchange graph, and
5. an algorithm for sorting N-numbers in $O(\log N)$ steps with high probability on an N-node shuffle-exchange graph with constant size queues.

Microsystems Research Center Room 39-321
Massachusetts Institute of Technology
Cambridge Massachusetts 02139
Telephone (617) 253-8138

Acknowledgements

To appear in the *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, May 1989. This research was supported in part by the Defense Advanced Research Projects Agency under contract number N00014-87-K-0825, the Office of Naval Research under contract number N00014-86-K-0593, the Air Force under contract number AFOSR-86-0076, and the Army under contract number DAAL-03-86-K-0171. Leighton: was supported in part by an NSF Presidential Young Investigator Award with matching funds from International Business Machines Corporation, Maggs: was supported in part by an NSF Graduate Fellowship, and Rosenberg: was supported in part by an NSF grant number CCR-88-12567.

Author Information

Koch: Mathematics Department and Laboratory for Computer Science, Room 2-333, MIT, Cambridge, MA 02139. (617) 253-7826.

Leighton: Mathematics Department and Laboratory for Computer Science, Room NE43-836A, MIT, Cambridge, MA 02139. (617) 253-5876.

Maggs: Laboratory for Computer Science, Room NE43-313, MIT, Cambridge, MA 02139. (617) 253-7843.

Rao: Laboratory for Computer Science, Room NE43-342, MIT, Cambridge, MA 02139. (617) 253-5889.

Rosenberg: Department of Computer Science, University of Massachusetts, Amherst, MA 01003.

Work-Preserving Emulations of Fixed-Connection Networks

(Extended Abstract)

Richard Koch[1,2] *Tom Leighton*[1,2] *Bruce Maggs*[2] *Satish Rao*[2] *Arnold Rosenberg*[3]

[1]Mathematics Department and
[2]Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

[3]Department of Computer Science
University of Massachusetts
Amherst, Massachusetts 01003

Abstract

In this paper, we study the problem of emulating T_G steps of an N_G-node guest network on an N_H-node host network. We call an emulation *work-preserving* if the time required by the host, T_H, is $O(T_G N_G/N_H)$ because then both the guest and host networks perform the same total work, $\Theta(T_G N_G)$, to within a constant factor. We say that an emulation is *real-time* if $T_H = O(T_G)$, because then the host emulates the guest with constant delay. Although many isolated emulation results have been proved for specific networks in the past, and measures such as dilation and congestion were known to be important, the field has lacked a model within which general results and meaningful lower bounds can be proved. We attempt to provide such a model, along with corresponding general techniques and specific results in this paper. Some of the more interesting and diverse conseqeuences of this work include:

1. a proof that a linear array can emulate a (much larger) butterfly in a work-preserving fashion, but that a butterfly cannot emulate an expander (of any size) in a work-preserving fashion,
2. a proof that a mesh can be emulated in real time in a work-preserving fashion on a butterfly, even though any $O(1)$-to-1 embedding of a mesh in a butterfly has dilation $\Omega(\log N)$,
3. a proof that an $N \log N$-node butterfly can be emulated in a work-preserving fashion on an N-node shuffle-exchange graph, and vice-versa,
4. simple $O(N^2/\log^2 N)$-area and $O(N^{3/2}/\log^{3/2} N)$-volume layouts for the N-node shuffle-exchange graph, and
5. an algorithm for sorting N-numbers in $O(\log N)$ steps with high probability on an N-node shuffle-exchange graph with constant size queues.

1 Introduction

1.1 The Problem

In this paper, we study the problem of emulating an N_G-node *guest* network $G = (V_G, E_G)$ on an N_H-node *host* network $H = (V_H, E_H)$ where $N_H \leq N_G$. Our goal is to emulate T_G steps of any computation on G in $T_H = ST_G$ steps on H where S (the *slowdown* of the emulation) is as small as possible.

The slowdown of the emulation must always be at least as large as N_G/N_H since G has N_G/N_H times as many processors as does H. If $S = O(N_G/N_H)$, then we say that the emulation is *work-preserving* because then the total *work* (i.e., the processor-time product) performed by the emulating network ($W_H = T_H N_H$) is within a constant factor of the work performed by the guest network ($W_G = T_G N_G$). Such emulations achieve optimal speedup (to within a constant factor) over sequential emulations of G since they use N_H processors to solve a problem $\Theta(N_H)$ times faster than is possible with a single processor.

More generally, we say that there is a *work-preserving emulation* of a class of networks $\mathcal{G}$ by a class of networks $\mathcal{H}$ with slowdown $S(N)$ if for every N and T, we can emulate any T steps of any $S(N)N$-node network in $\mathcal{G}$ in $O(S(N)T)$ steps on any N-node network in $\mathcal{H}$. If $S(N) = O(\log^\alpha N)$ for some constant α, then we say that the emulation is *NC work-preserving* since every step of G can be emulated in $O(\log^\alpha N)$ steps of H. If $S(N) = O(N^\alpha)$ for some constant α, then we say that the emulation is *polynomial time work-preserving*, and so on. In the special case that $S(N) = O(1)$, we say that the emulation is *real-time*. Real-time emulations are the hardest to obtain since we require the host network to emulate a guest network of the same size with constant slowdown.

As a simple example, let $\mathcal{G}$ be the class of linear arrays, and $\mathcal{H}$ be the class of all bounded-degree connected

This research was supported by the Defense Advanced Research Projects Agency under Contract N00014-87-K-825, the Office of Naval Research under Contract N00014-86-K-0593, the Air Force under Contract OSR-86-0076, and the Army under Contract DAAL-03-86-K-0171. Tom Leighton is supported by an NSF Presidential Young Investigator Award with matching funds provided by IBM. Bruce Maggs is supported by an NSF Graduate Fellowship. Arnold Rosenberg is supported by NSF Grant CCR-88-12567.

networks. It is well known [18] that an N-node linear array can be embedded one-to-one in any connected bounded-degree N-node network with constant dilation and congestion. (By an embedding of a graph G into a graph H, we mean a mapping $\phi : G \rightarrow H$ that maps the nodes of G to the nodes of H and the edges of G to paths in H. The *dilation* of an embedding is the length of the longest path $\phi(e)$ corresponding to an edge of G. The *congestion* of an embedding is the largest number of paths $\phi(e)$ crossing a single edge of H. The *load* of an embedding is the maximum number of nodes of G mapped to a single node of H. In a one-to-one embedding, the load is 1.) Hence any N-node bounded degree connected network H can emulate any N-node linear array with constant slowdown, and thus there is a real-time emulation of the class $\mathcal{G}$ by the class $\mathcal{H}$.

As another simple example, consider the more interesting problem of emulating a butterfly on a linear array. We will prove that the class of butterflies cannot be real-time emulated by the class of linear arrays. (This should come as no surprise, although the proof is not entirely trivial.) However, there is a simple work-preserving emulation of the class of butterflies by the class of linear arrays with slowdown 2^N. In particular, consider an $N2^N$-node butterfly with nodes and edges

$$V = \{\langle i, w\rangle | 1 \leq i \leq N, w \in \{0,1\}^N\}, \text{ and}$$

$$E = \{(\langle i, w\rangle, \langle i', w'\rangle) | i' = i+1, w' = w \text{ or } w' = w^{(i)}\},$$

where $w^{(i)}$ denotes w except that the ith bit is changed. Then by mapping the 2^N nodes of the form $\langle i, w\rangle$ (where $w \in \{0,1\}^N$) to the ith node of the linear array, an N-node linear array can emulate an $N2^N$-node butterfly with 2^N slowdown.

Seeing this elementary example, one is tempted to ask if there are faster work-preserving emulations of a butterfly on a linear array. In other words, can we emulate a smaller butterfly (say with polynomial blowup) in a work-preserving fashion on a linear array? Although the proof is not obvious, the answer is no. There is no polynomial-time work preserving emulation of the class of butterflies by the class of linear arrays. Any such emulation requires exponential slowdown. Alternatively, we might wonder if a linear array can emulate any bounded-degree network in a work-preserving fashion given enough slowdown. Again, the answer is no. Although the linear array can emulate a butterfly in a work-preserving fashion, it cannot emulate any expander, no matter how much blowup is allowed. In fact, by combining these results we can conclude that even a butterfly is not sufficiently powerful to emulate an expander in a work-preserving fashion.

We also consider emulations that are not work-preserving. Such emulations are (by definition) inefficient, and we define the inefficiency of such an emulation to be $I = W_H/W_G$. In these terms, an emulation is work-preserving if it has constant inefficiency. Many of our bounds will reflect tradeoffs between slowdown and inefficiency. In general,

$$I = \frac{S}{C}$$

where $C = N_G/N_H$ is the *contraction* of an emulation.

1.2 The motivation

There are several good reasons for studying the problem of emulating one network on another in a work-preserving fashion. For starters, this kind of analysis gives us an excellent means by which to compare the computational power of one network relative to that of another. More importantly, it gives us an automatic way to compile and run algorithms designed for one kind of parallel architecture without loss of efficiency on another. This is provided, of course, that the ratio of the size of the problem to the size of the machine is large enough. For example, we have already seen that a small linear array (which has a very simple structure) is just as efficient in terms of work as a very large butterfly (which has a more complicated structure).

More generally, the study of work-preserving emulations lies at the heart of efficient parallel computing. Indeed, one of the central problems in efficient parallel computing is the task of mapping a collection of processes linked by precedence and/or communication constraints onto the processors and routing network of a parallel machine so that

1. the processing load imposed on the processors is balanced,

2. the communication between processors can be handled efficiently, and

3. the computation and communication can be scheduled so that the necessary inputs for a process are available where and when the process is scheduled to be computed.

In other words, we would like to schedule the communication and computation in a way that takes maximum advantage of the available hardware to minimize the completion time of the job.

In general, we can model the computation to be performed by a DAG. Each node of the DAG represents a process and each directed edge (u, v) represents a communication that must take place between u and v. Typically, this communication represents data output from u after u is completed which is to be input to v before v is started. The parallel machine can be modeled as an undirected network. The nodes of the network correspond to processors, and the edges correspond to communication links between processors (and/or their associated memories). The implementation of the computation to be performed on the parallel machine then corresponds to an embedding of the DAG in the network so that nodes of the DAG are mapped to nodes of the network and so that edges of the DAG are mapped to paths in the network. We may also need to construct a schedule that specifies the communication and computation of the DAG that is being performed during each step of the network. This will be particularly important if the parallel machine is synchronous.

In many applications, the DAG possesses a very natural structure. For example, typical DAGs encountered in practice are derivitives of a binary tree, array, butterfly, or shuffle-exchange graph. This is often due to the fact that the DAG is associated with an algorithm whose inherent underlying structure is a tree or array (as is the case for many problems in numerical analysis and linear algebra) or a butterfly or shuffle-exchange graph (as is the case for Fourier Transform and data manipulation problems). Alternatively, it could be that the DAG was constructed from an algorithm specifically designed for use on one of these common parallel architectures.

Similarly, parallel networks also tend to be very naturally structured and typically are configured as trees, arrays, butterflies, and the like. Hence, the mapping problem often consists of emulating T_G steps of one N_G-node network (represented as a $T_G N_G$-node DAG) on an N_H-node network with a different structure. Ideally, we would like to perform the computation in $O(T_G N_G/N_H)$ steps, which is precisely the problem of finding a work-preserving emulation of one network on another.

In practice, the guest network can be substantially larger than the host network. For example, it is not uncommon for a parallel machine with between 8 and 256 processors to be emulating array-based computations involving hundreds of thousands of data points. In such examples, even work-preserving emulations with exponential slowdown may be within the scope of practicality. Indeed, the most important feature of the computation is that it be work-preserving. In fact, the notion of a work-preserving computation is important enough that it transcends high-level architectural issues such as SIMD vs. MIMD, synchronous vs. asynchronous, small scale vs. large scale, and fine grain vs. coarse grain. For example, even though issues involving the timing of computations and communications become muddied with asynchronous architectures, the underlying problem of embedding the computation so as to mimimize computational load and communication load (independent of timing) still remains. As a consequence, work-preserving emulations are just as important for a Dataflow Machine as they are for a Connection Machine (to mention two architectures at opposite ends of the spectrum).

1.3 A closer look at the computational model

If we can find an embedding of a graph G into a graph H with constant dilation, congestion, and load, then it is fairly clear that H can emulate G with constant slowdown. Is the reverse true? Somewhat surprisingly, it is not. For example, Bhatt, Chung, Hong, Leighton and Rosenberg [2] proved that any embedding of an N-node mesh into an N-node butterfly with constant load requires dilation $\Omega(\log N)$, the worst possible. At first glance, it might seem that this result implies that there is no real-time emulation of a mesh on a butterfly. As we show in this paper, however, this is not the case. There is, in fact, a way to emulate T steps of an N-node mesh computation in $O(T)$ steps on an N-node butterfly for any T.

In order to understand how such a contradictory result is possible, we need to take a closer look at what it means to emulate T_G steps of one network in T_H steps on another. We start by modeling the computation performed by the guest network G as a pebble DAG Γ. In particular, we will have a pebble for every node-time pair (v,t) where v is a node of G and $0 \le t \le T_G$. (Pairs of the form $(v,0)$ correspond to inputs.) In fact, we may have many pebbles associated with a single pair (v,t), which will correspond to the same computation being done more than once. (This is the trick that allows us to emulate a mesh on a butterfly in real time.) To compute any pebble labeled (v,t), we need as inputs pebbles labeled $(v,t-1)$ and $(v_1,t-1),(v_2,t-1),\ldots,(v_k,t-1)$, where $v_1,v_2,\ldots,v_k$ are the neighbors of v in G. We use the directed edges of Γ to denote this dependence in the usual way.

Because many pebbles can have the same label, there are many DAGs Γ associated with any graph G. In order to emulate G on H, we only need to find an embedding and an acompanying schedule of one of these DAGs in H. Once an embedding and schedule of a DAG is fixed, the emulation proceeds in a standard way. In particular, during each step of the computation, a node of H can

1. make a copy of a single pebble that it contains,
2. send a single pebble to a neighbor, and/or
3. create a pebble with label (v,t) provided that it already contains input pebbles with labels $(v,t-1)$ and $(v_1,t-1),(v_2,t-1),\ldots,(v_k,t-1)$.

Initially, we will allow a node of H to have access to any input, although to use any of these inputs in a meaningful way will take time. By the end of the emulation, we must have computed pebbles with all labels of the form (v,T_G). (For purposes of simplicity, we will use a pebble to denote the state of a processor of G at some particular time, as described above. A more general interpretation would be to use a pebble to denote one of many items (e.g., data and/or functions) stored within a processor. All of our results hold under the more general interpretation, although some of the emulation results become more complicated.)

By allowing several pebbles to have the same label, we dramatically increase the number of possible computation DAGs Γ that correspond to a T_G-step computation of G. This makes it more likely that we can find a computation that can be efficiently emulated on some host network H (e.g., as is the case with emulating a mesh on a butterfly), but it also makes the task of proving lower bounds much more difficult. For example, in order to prove that H cannot emulate G in real-time, we must show that for some T_G, there is no DAG Γ associated with a T_G-step computation of G that can be emulated in $O(T_G)$ steps on H. This can be a formidable task since Γ can look very different than G. Indeed, at the very least, we must choose T_G to be large since by allowing redundant computations of pebbles, any $O(1)$ steps of any N-node bounded-degree graph G can be computed in $O(1)$ steps on any N-node graph H. (This is because if $T = O(1)$, then any output pebble can only depend on $O(1)$ input pebbles, which can be redundantly computed locally since every node of H is assumed to have access to all input pebbles.)

Note that when we prove a lower bound on the ability of a graph H to emulate a graph G, it does not necessarily mean that H cannot effectively compute the same result as does G (possibly by using a different algorithm, for example). Rather, we are proving lower bounds on the ability of H to perform the same step-by-step computations as G when G is used in a general purpose way. Hence the term *emulation*. We suspect that our pebbling model is probably the most general model in which we could hope to prove lower bounds.

Throughout the paper we will make use of the fact *that if there is an embedding of G in H with congestion* c, dilation d, and load l, then there is an emulation of G by H with slowdown $O(l + c + d)$. This follows for any H from the construction in [11]. When H is an array, tree, butterfly, or shuffle-exchange graph, the schedule can be computed on-line using the randomized routing algorithm in [11].

1.4 Our results

The technical portion of this paper is divided into five sections. We commence in Section 2 with some general techniques for establishing the existence or nonexistence of a work-preserving emulation. In particular, we describe two general methods for proving lower bounds on the slowdown of a work-preserving emulation. The first method is based on dilation considerations and appears in Section 2.1. As an application of this method, we prove that any class of low diameter networks (such as complete binary trees) cannot be emulated in real time on any class of networks that has poor expansion properties (such as arrays of bounded dimension).

The second method is based on congestion properties and is presented in Section 2.2. Here we describe a general method for proving that a work-preserving emulation requires a large amount of time, or that it is impossible altogether. As an example, we prove that any work-preserving emulation of a butterfly on an array of bounded-dimension requires exponential time, and that it is not possible to emulate an expander on a butterfly in work-preserving fashion. These results provide a curious contrast between the power of a linear array, butterfly, and an expander. By most standards, it would seem that a butterfly is closer in power to an expander than it is to a linear array. Yet a linear array can emulate a butterfly in a work-preserving fashion, but a butterfly (or most any non-expander) cannot emulate an expander in a work-preserving fashion.

In Sections 3-6 of the paper, we focus on the special case of emulations on arrays, complete binary trees, butterflies, and shuffle-exchange graphs, respectively. In Section 3, we prove tight bounds on the slowdown required for an array to emulate a tree, array or butterfly. In Section 4, we prove that there is a work-preserving emulation of bounded-degree trees by complete binary trees with $O(\log\log N)$ slowdown. We also give evidence, but no proof, that there is no corresponding real-time emulation for this class. (Proving that a complete binary tree cannot emulate a complete ternary tree in real-time is one of several challenging questions left open in this paper.)

In Section 5, we show that the class of arrays with bounded dimension can be emulated in real-time on a butterfly. This result is interesting because any one-to-one embedding of an array (with dimension 2 or more) in a butterfly requires $\Omega(\log N)$ dilation [2], which suggests that a real-time emulation is not possible. The result takes on added significance given the fact that many parallel numerical algorithms are array-based while several parallel machines are butterfly-based.

We also describe a simple constant-congestion embedding of an N-node shuffle-exchange graph in an N-node butterfly in Section 5. This result has several important consequences. First, it can be used to provide an elementary proof that the N-node shuffle-exchange graph can be laid out in $O(N^2/\log^2 N)$ area and in $O(N^{3/2}/\log^{3/2} N)$ volume. Both results are optimal. The area bound was known previously [7], but the proof was much more difficult (as were the proofs for several nonoptimal layouts for the shuffle-exchange graph [6, 10, 12, 19]). The 3-d layout bound is new and was not obtainable by any of the previous approaches to the 2-d layout problem. Second, we apply the result to derive an $O(\log N)$-slowdown work-preserving emulation of the shuffle-exchange graph on the butterfly.

In Section 6, we prove the reverse, namely, that there is an $O(\log N)$-slowdown work-preserving emulation of the butterfly on the shuffle-exchange graph. Taken together, these results come very close to resolving a long open question concerning whether or not the butterfly and shuffle-exchange graph are computationally equivalent. In particular, we show that up to NC emulations, the butterfly and shuffle-exchange graphs are equivalent in a work-preserving sense. Thus, for many problems, they can be considered to be computationally equivalent.

As a consequence of the emulations in Section 6, we also obtain a real-time emulation of bounded-degree arrays in the shuffle-exchange graph, and we show how to sort N numbers with high probability in $O(\log N)$ steps on an N-node shuffle-exchange graph. Although the proof of the sorting bound is elementary, it resolves an open question concerning the difficulty of randomized sorting algorithms on the shuffle-exchange graph. Previously, such an algorithm was known for the butterfly [11, 15, 17] but that algorithm made crucial use of the recursive structure of the butterfly, a structure not present in a shuffle-exchange graph.

1.5 Previous work

There has been a great deal of previous work on graph embeddings with the intent of showing that one network can or can't emulate another network efficiently [2, 3, 4, 5, 11, 16]. Many of the results were positive and proved things like "all N-node binary trees can be emulated in constant time on an N-node hypercube." There were also some negative results, but because of the lack of a good model, their significance is now less clear. For example, even though an embedding of a mesh into a butterfly requires dilation $\Omega(\log N)$, we now find that a butterfly can emulate a mesh with constant slowdown.

The notion of work-preserving emulations in PRAM models has previously been studied [8, 13] and served to motivate this work. Related problems of scheduling computations on fixed-connection networks have also been studied [14].

2 Lower bounds

In this section we present lower bounds on slowdown and inefficiency. Loosely speaking, these lower bounds apply when the guest graph expands faster than the host graph. The first lower bound can be used to show that any emulation of a complete binary tree by a linear array has slowdown $\Omega(N_H/\log N_H)$. The second can be used to show that a butterfly cannot perform a work-preserving emulation of an expander graph, that any work-preserving emulation of a butterfly by a linear array H requires slowdown at least $2^{\Omega(N_H)}$, and that any work-preserving emulation of a $k+1$-dimensional mesh by a k-dimensional mesh H requires slowdown at least $\Omega(N_H^{1/k})$. All of these lower bounds on slowdown are tight.

Before proving the lower bounds, we need to introduce some notation. For an undirected graph $G = (V, E)$, let $\delta(u,v)$ be the length (number of edges) of the shortest path between nodes u and v in G. Let $B_G(u,i) = \{v \in V | \delta(u,v) \leq i\}$ be the set of nodes within a distance i of u in G and let $b_G(u,i) = |B_G(u,i)|$. We call b_G the growth function of G.

2.1 Distance-based lower bound

The following theorem shows that if the guest graph grows faster than the host graph, then any emulation of the guest by the host must be slow.

Theorem 1 *Let $H = (V_H, E_H)$ be an N_H-node host graph and $G = (V_G, E_G)$ be an N_G-node guest graph, and suppose that there are integers τ_H and τ_G such that*

$$\max_{u \in V_H} \sum_{i=1}^{\tau_H} b_H(u,i) < \min_{v \in V_G} \sum_{j=1}^{\tau_G} b_G(v,j).$$

Then any emulation of $T_G \geq \tau_G$ steps of G by H has slowdown

$$S > (\tau_H + 1)/2\tau_G.$$

Proof: The basic idea is to find a sequence of T_G/τ_G pebbles in any T_G-step pebble DAG of G such that each pair of pebbles is separated by at most τ_G guest time steps but are created in H at least τ_H host time steps apart. As we shall see, such a sequence exists only if the slowdown $S = T_H/T_G$ is at least $(\tau_H+1)/2\tau_G$.

We start the sequence with the last pebble created by H. Suppose that at time T_H some node $u_0 \in V_H$ creates a pebble for DAG node (v_0, t_0), where $t_0 = T_G$. The pebble for (v_0, t_0) cannot be created by H until pebbles for all of its predecessors in the DAG are created. In particular, there are at least $\sum_{j=1}^{\tau_G} b_G(v_0, j)$ predecessors for time steps $t_0 - \tau_G$ through $t_0 - 1$. We want to show that the pebble for at least one of these predecessors must have been created by the host graph before time $T_H - \tau_H$. The pebble for every predecessor of (v_0, t_0) that is created at distance i from u_0 in H must be created at or before time $T_H - i$. Thus at most $\sum_{i=1}^{\tau_H} b_H(u_0, i)$ pebbles for predecessors of (v_0, t_0) are created by H between time steps $T_H - \tau_H$ and $T_H - 1$. Since $\max_{u\in V_H} \sum_{i=1}^{\tau_H} b_H(u,i) < \min_{v \in V_G} \sum_{j=1}^{\tau_G} b_G(v,j)$, the pebble for some predecessor (v_1, t_1), $t_1 \geq T_G - \tau_G$, must be created by the host graph at or before time $T_H - (\tau_H + 1)$.

We can repeat the argument to find a pebble for a predecessor (v_2, t_2), $t_2 \geq T_G - 2\tau_G$, of (v_1, t_1) that must be created by the host at or before time $T_H - 2(\tau_H + 1)$, and so on. Eventually we obtain a pebble (v_k, t_k) such that $\tau_G > t_k \geq T_G - k\tau_G$. This pebble must be created by the host at or before time $T_H - k(\tau_H+1)$. We assume that input pebbles are created at host time step 0, and that the emulation begins with time step 1. Thus, $T_H - k(\tau_H + 1) \geq 0$. Combining these inequalities, we have

$$T_H/T_G > (\tau_H + 1)/2\tau_G$$

for $T_G \geq \tau_G$. □

Corollary 2 *Any such emulation has inefficiency*

$$I > \Omega\left(\frac{\tau_H N_H}{\tau_G N_G}\right).$$

Corollary 3 *Any emulation of a complete binary tree, G, by a k-dimensional mesh, H, has slowdown at least $\Omega\left((N_G/\log^k N_G)^{1/(k+1)}\right)$.*

Proof: Apply Theorem 1 with $\tau_G = \Theta(\log N_G)$, and $\tau_H = \Theta\left((N_G \log N_G)^{1/(k+1)}\right)$. □

2.2 Congestion-based lower bound

The second lower bound requires a little more notation. Let $G = (V, E)$ be an undirected graph as before. For a set $U \subseteq V$, we define the i-neighborhood of U to be the set of nodes within a distance i of some node in U, $\mathcal{N}_i(U) = \cup_{u \in U} B_G(u, i)$. We define an $(R, f(R))$-decomposition of G to be a partition of V into $|V|/R$ sets of nodes (regions) such that each contains R nodes and has a 1-neighborhood of size at most $f(R)$.

The last graph parameter that we need, z_G, is best described in terms of a simple game. The player starts by choosing a nodes of a connected graph G and placing them in a bag. The player is given a collection of εa, $0 \leq \varepsilon < 1$, tokens to play with. The game is played in rounds, each consisting of two steps. In the first step, all of the neighbors of the nodes in the bag are added to the bag. In the second step, the player may exchange tokens for nodes in the bag on a one-for-one basis. Let X_i be the set of nodes in the bag at the end of round i, and let Y_i be the set of nodes removed in the second step of round i. Then X_i is given by the recurrence $X_i = \mathcal{N}_1(X_{i-1}) - Y_i$. The game ends when the number of nodes in the bag exceeds it capacity, c, at the end of a step, where $c < N_G$. If k is the number of rounds played, then $|X_i| \leq c$ for $i < k$, $|X_i| > c$ for $i = k$, and $\sum_{i=1}^{k} |Y_i| \leq \varepsilon a$. The goal is to play as many rounds as possible. Let $z_G(a, \varepsilon, c)$ be an upperbound that is non-increasing in a on the length of the longest possible game.

Theorem 4 *Suppose that $H = (V_H, E_H)$ is an N_H-node host graph with an $(R, f(R))$-decomposition, and that $G = (V_G, E_G)$ is an N_G-node guest graph. Let*

$$\beta = \max\left\{ z_G\left(\frac{N_G}{4}, 0, \frac{3N_G}{4}\right), z_G\left(\frac{3N_G R}{8N_H}, \frac{1}{2}, \frac{N_G}{2}\right)\right\}.$$

Then for any emulation of G by H where $T_G > 3\beta$,

$$I \geq \min\left\{\frac{R}{32\beta f(R)}, \frac{N_H}{96R}\right\}.$$

Proof: The basic strategy is to show that either the host spends a lot of time passing pebbles across the perimeters of the regions in the $(R, f(R))$-decomposition, or the host spends a lot of time creating pebbles. We will break the T_G guest time steps into blocks of 3β consecutive steps and classify every block as either an *importer* or a *creator*. If a block is an importer, then many pebbles for the block cross region perimeters. If a block is a creator, then some region creates many pebbles for the block. If the majority of the blocks are importers, then the time required by the host to pass pebbles across the perimeters of the regions large. Otherwise, the time required to create the pebbles is large.

Before we can get started we need one more piece of notation. For each node v in G there is at least one pebble created by H for each guest time step t between 1 and T_G. The first pebble created for v for time t is called the *t-primary pebble* for v. For each value of t there are exactly N_G t-primary pebbles. The t-primary pebbles are ordered according to the order in which they are created by H, with ties broken arbitrarily. We call the first $3N_G/4$ t-primary pebbles the *t-early* pebbles and the last $3N_G/4$ the *t-late* pebbles.

We begin with the definition an importer block. Consider a block from step t to $t - 3\beta + 1$. The average number of t-early pebbles created by each of the N_H/R regions in the decomposition of H is at least $p = 3N_G R/4N_H$. We say that a region is *t-busy* if it creates at least $p/2$ t-early pebbles. We say that a t-early pebble is t-busy if it is created by a t-busy region. At least half of the t-early pebbles are t-busy. Thus, there are at least $3N_G/8$ t-busy pebbles. Suppose that a t-busy region creates $s \geq p/2$ t-busy pebbles. We say that the region is an *importer* if it imports at least $s/2$ *pebbles for time steps between* $t-1$ and $t - 2\beta$. We say that a block is an importer if every t-busy region is an importer, or if some region imports at least $3N_G/16$ pebbles for time steps between $t - 1$ and $t - 2\beta$. In a importer block, a total of at least $3N_G/16$ pebbles for time steps between $t-1$ and $i - 2\beta$ are imported by all of the regions.

If at least half of the $T_G/3\beta$ blocks are importers, then we can find a lower bound on inefficiency by computing the time required to import pebbles. In this case, the total number of pebbles imported by all of the importer blocks is at least $T_G N_G/32\beta$. The host time required to import these pebbles is at least $T_H \geq T_G N_G R/32\beta N_H f(R)$, because at each host time step, each of the N_H/R regions can import at most $f(R)$ pebbles. In this case,

$$I \geq R/32\beta f(R).$$

As we shall see, if a block is not an importer then some region must create many pebbles for the block. Hence the name creator. In a creator block there must be some t-busy region $\mathcal{R}$ that creates $s \geq p/2$ t-busy pebbles but imports fewer than $s/2$ pebbles for time steps between $t - 1$ and $t - 2\beta$. The t-busy pebbles created by $\mathcal{R}$ cannot be created until pebbles for all of their predecessors in the pebble DAG are created. Since $z_G(s, 1/2, N_G/2) \leq z_G(p/2, 1/2, N_G/2) \leq \beta$, $\mathcal{R}$ imports at most $s/2$ pebbles for time steps between $t-1$ and

$t-z_G(s,1/2,N_G/2)$. Thus $\mathcal{R}$ must create at least $N_G/2$ pebbles for time step $t-z_G(s,1/2,N_G/2)$. Furthermore, since $\mathcal{R}$ imports at most $3N_G/16$ pebbles for time steps between $t-1$ and $t-2\beta$, it must create at least $5N_G/16$ pebbles for every time step between $t-z_G(s,1/2,N_G/2)$ and $t-2\beta$. For each of these time steps, at least $N_G/16$ of the pebbles are created for nodes whose $(t-2\beta)$-primary pebbles are $(t-2\beta)$-late pebbles. We call these pebbles the *descendant* pebbles.

We have chosen the descendant pebbles so that none are created by H until all of the descendant pebbles for previous blocks have been created. The early pebbles for all time steps at or before $t-2\beta-z_G(N_G/4,0,3N_G/4)$ must be created before the $(t-2\beta)$-late pebbles because $3N_G/4$ nodes in G lie within a distance $z_G(N_G/4,0,3N_G/4)$ of the nodes corresponding to the first $N_G/4$ $(t-2\beta)$-primary pebbles. Since $z_G(N_G/4,0,3N_G/4)\le\beta$, the early pebbles for previous blocks must be created before the $(t-2\beta)$-late pebbles. Furthermore, the $(t-2\beta)$-late pebbles must be created before the descendant pebbles, which in turn must be created before the t-busy pebbles for $\mathcal{R}$.

If at least half of the blocks are creators, then we can derive a lower bound on inefficiency by summing the time to create the descendant pebbles for each of the creator blocks. For each of $T_G/6\beta$ creator blocks, at least $\beta N_G/16$ descendant pebbles are created by a single region. The host time for each block is at least $\beta N_G/16R$. The host time for all of the creator blocks is at least $T_G N_G/96R$ and the inefficiency is at least

$$I\ge N_H/96R.$$

Combining the two cases proves the theorem. □

Corollary 5 *A k-dimensional mesh H cannot perform a work-preserving emulation of an expander graph G.*

Proof: Apply Theorem 4 with $R=\Theta((N_H\log N_H)^{k/(k+1)})$, $f(R)=O(R^{(k-1)/k})$, and $\beta=O(\log(N_H/R))$. The inefficiency is at least $I\ge\Omega((N_H/\log^k N_H)^{1/(k+1)})$. □

Corollary 6 *A butterfly network H cannot perform a work-preserving emulation of an expander graph G.*

Proof: Apply Theorem 4 with $R=\Theta(\sqrt{N_H}\log N_H)$, $f(R)=O(\log R)$, and $\beta=O(\log(N_H/R))$. The inefficiency is at least $\Omega(\sqrt{N_H}/\log N_H)$. □

Corollary 7 *Any work-preserving emulation of a butterfly G by a k-dimensional mesh H has slowdown at least $2^{\Omega(N_H^{1/k})}$.*

Proof: Apply Theorem 4 with $R=\Theta((N_H\log N_G)^{k/(k+1)})$, $f(R)=O(R^{(k-1)/k})$, and $\beta=O(\log N_G)$. The inefficiency is at least $I\ge\Omega((N_H/\log^k N_G)^{1/(k+1)})$. □

Corollary 8 *Any work-preserving emulation of a j-dimensional mesh G by a k-dimensional mesh H, $j>k$, has slowdown at least $\Omega(N_H^{(j-k)/k})$.*

Proof: Apply Theorem 4 with $R=\Theta((N_G^{1/j}N_H)^{k/(k+1)})$, $f(R)=O(R^{(k-1)/k})$, and $\beta=O(N_G^{1/j})$. The inefficiency is at least $I\ge\Omega((N_H^j/N_G^k)^{1/j(k+1)})$. □

3 Emulations by arrays

Although the arrays cannot perform real-time emulations of graphs with small diameter, we can show that they can perform work-preserving emulations of complete binary trees, other arrays, and butterflies. In each case, we find an embedding of the guest graph into the array with acceptable load, congestion, and dilation. The edges of the guest graph are emulated by routing packets between the nodes of the linear array. All of the following results can be shown to be tight by Corollaries 3, 8, and 7.

Observation 9 *An N-node k-dimensional mesh can perform a work-preserving emulation of an $N^{(k+1)/k}/\log N$-node complete binary tree.*

Proof: An $N^{(k+1)/k)}/\log N$-node complete binary tree can be embedded in an N-node k-dimensional mesh with load $O(N^{1/k}/\log N)$, dilation $O(N^{1/k}/\log N)$, and congestion $O(N^{1/(k+1)})$. □

Observation 10 *An N-node k-dimensional mesh can perform a work-preserving emulation of an $N^{j/k}$-node j-dimensional mesh, $j>k$.*

Proof: An $N^{j/k}$-node j-dimensional mesh can be embedded in an N-node k-dimensional mesh with load $N^{(j-k)/k}$, congestion $N^{(j-k)/k}$, and dilation 1. □

Observation 11 *An $N_H=n^k$-node k-dimensional mesh H can perform a work-preserving emulation of an $N_G=n2^n$-node butterfly graph G.*

Proof: An $n2^n$-node butterfly graph with 2^n rows and n columns can be embedded in a $N_H=n^k$-node k-dimensional mesh with load $O(2^n/n^{k-1})$, congestion $O(2^n/n^{k-1})$, and dilation $O(n)$. □

It is interesting to note that every connected network can perform a real-time emulation of a linear array. Hence, Observations 9 through 11 can be modified to hold for all connected networks.

4 Emulations by complete binary trees

4.1 Work-preserving emulations of bounded-degree trees

In this section, we show that any $N\log\log N$-node forest with maximum degree Δ can be embedded in an

N-node complete binary tree with load $O(\Delta \log\log N)$, congestion $O(\Delta^2 \log\log N)$, and dilation $O(\log \Delta)$. As a corollary, there is a work-preserving emulation with slowdown $O(\log\log N)$ of the class of bounded-degree forests by the class of complete-binary trees.

In constructing the embedding, we use the following weighted separator theorem for forests.

Theorem 12 *Suppose that $F = (V, E)$ is a forest where each vertex has been assigned some non-negative weight. Then it is possible remove a set S of k vertices such from V such that the remaining vertices can be partitioned into two subforests F_1 and F_2 such that no edge connects a vertex in F_1 with a vertex in F_2, and each contains at most $|V|(1+(2/3)^{k/2})/2$ vertices and at most $5/6$ of the total weight.*

Proof: Omitted.

We begin by using Theorem 12 to find a set S of $k \in O(\log\log N)$ nodes that partitions the forest $F = (V, E)$ into two subforests, each containing at most $|V|(1 + 1/\log N)/2$ vertices. We embed S at the root of the binary tree and then recursively embed one of the subforests in the left subtree of the root, and the other in the right.

At levels below the root, we use Theorem 12 to simultaneously partition the vertices of the forest and the edges connecting the forest to vertices that are embedded higher in the binary tree. Let $F_i = (V_i, E_i)$ be a forest to be embedded in a subtree rooted at a level i node v_i in the binary tree. Let N_i be the number of edges connecting F_i to vertices embedded higher in the binary tree; N_i is the congestion of of the binary tree edge connecting v_i to its parent. We assign each vertex of F_i a weight equal to the number of neighbors it has that are embedded higher in the binary tree. Using Theorem 12, we find a set S_i of k vertices that partitions F_i into two subforests, each of size at most $|V_i|(1+1/\log N)/2$, and each having at most $(5/6)N_i$ edges to vertices that are embedded higher in the tree. We embed the vertices of S_i at v_i and recursively embed one of the subforests in the left subtree of v_i, and the other in the right subtree.

To limit the dilation to some integer d, whenever i is a multiple of d we embed at v_i not only S_i but also all of the vertices in F_i that have at least one neighbor embedded somewhere higher in the binary tree.

We must now show how to choose d so that both the congestion and the load of the embedding are small. Consider any simple path from a level i node v_i in the binary tree to a level $i+d$ node, v_{i+d}, where i is a multiple of d. At level i, we embed a separator of size k and at most N_i other vertices that have at least one neighbor embedded higher in the tree. Since each of these vertices has at most Δ neighbors, $N_{i+1} \leq \Delta k + \Delta N_i$. At level $i+1$, we embed a separator of size k that partitions F_{i+1} into two subforests, each having at most $(5/6)N_{i+1}$ edges to vertices embedded higher in the binary tree. Thus, at level $i+2$, we have $N_{i+2} \leq (5/6)N_{i+1} + \Delta k$. In general, N_{i+j} is given by the recurrence

$$N_{i+j} \leq \begin{cases} \Delta k + \Delta N_i & j = 1 \\ (5/6)N_{i+j-1} + \Delta k & 1 < j \leq d \end{cases}$$

Solving the recurrence yields

$$N_{i+j} \leq 6\Delta k + (5/6)^{j-1}\Delta N_i.$$

We are now in a position to calculate the load and the congestion. The preceeding argument shows that for $d \in O(\log \Delta)$ and $N_i \in O(\Delta k)$, we have $N_{i+d} \leq N_i$. Thus, in every simple path between a node at level i and a node at level $i+d$, where i is a multiple of Δ, the congestion starts at $O(\Delta k)$ at level i, rises to at most $O(\Delta^2 k)$ at level $i+1$ and proceeds to drop back down to at most $O(\Delta k)$ at level $i+d$. Thus, the congestion of the embedding is at most $O(\Delta^2 \log\log N)$. How large can the load be? At each node of the binary tree we embed a separator of size k. For every i that is a multiple of d, we also embed a set nodes of size $N_i = O(\Delta k)$. Finally, at the leaves we embed forests of size $N \log\log N((1+1/\log N)/2)^{\log N}$, which is at most $O(\log\log N)$. Thus the load is at most $O(\Delta \log\log N)$.

4.2 Congestion lower bounds for a complete ternary tree

In this section we show that any embedding of an N-node complete ternary tree in an N-node complete binary tree with load at most $O(\sqrt{\log\log N})$ in which the leaves of the ternary tree are mapped to the leaves of the binary tree has congestion at least $\Omega(\sqrt{\log\log N})$. This lower bound suggests, but does not prove, that real-time emulation of a complete ternary tree by a complete binary tree is impossible.

Theorem 13 *Any embedding of an N-node complete ternary tree in an N-node complete binary tree with load at most $O(\sqrt{\log\log N})$ in which the leaves of the ternary tree are mapped to the leaves of the binary tree has congestion at least $\Omega(\sqrt{\log\log N})$.*

Proof: Omitted.

5 Emulations in a butterfly graph

Before describing our emulations we give some notation concerning the butterfly graph. Recall that a butterfly graph node can be represented by a pair $< i, w >$. We refer to i as the node's *level*. We refer to w as the node's *position in level (PIL)*. We consider the nodes of the butterfly with the same PIL to be in a *row*. We consider the inputs of the butterfly to be the nodes whose representatives are of the form $< 0, w >$, i.e., the level 0 node of a row. In the following sections we will connect the inputs of a butterfly to each other via paths through the butterfly. We make use the following theorem of Benes [1].

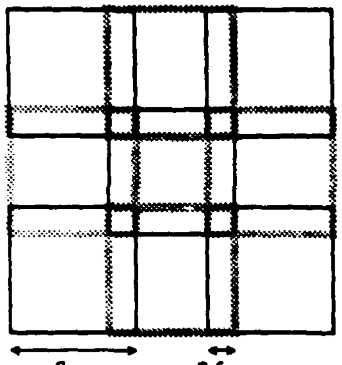

Figure 1: Division of the mesh into submeshes

Theorem 14 *The inputs of an $N \log N$-node butterfly can be connected in any permutation by a set of paths such that each path has length at most $2 \log N$, and each edge in the butterfly is used at most twice (once in each direction).*

5.1 Work-preserving emulations of binary trees

When the Bhatt, Chung, Hong, Leighton, Rosenberg result [2] that a butterfly can emulate a complete binary tree in real-time is combined with the material in Section 3, we find that there is an $O(\log \log N)$-time work-preserving simulation of the class of binary trees on the butterfly. Whether or not this emulation can be performed in real-time remains an open question.

5.2 Real time emulation of arrays

Theorem 15 *For constant q, T steps on a $\sqrt[q]{N} \times \cdots \times \sqrt[q]{N}$ q dimensional mesh can be emulated in $O(T)$ steps on a butterfly graph with $O(N)$ nodes.*

Proof. We prove the theorem for $q = 2$; for other values of q the proof is similar. We will only prove the theorem when $T \geq \log N$; when $T < \log N$ the proof is similar.

We will prove the theorem using recursion. We will divide the mesh into submeshes and the butterfly into subbutterflies and recursively emulate each submesh in a subbutterfly. Since submeshes will need pebbles computed in other submeshes, we will create connections between the submeshes.

Suppose that we know how to emulate f_{k+1} steps of a $s_{k+1} \times s_{k+1}$ mesh on a butterfly with $N_{k+1} = n_{k+1}2^{n_{k+1}}$ nodes, n_{k+1} is a power of two, and $s_{k+1}^2 = m_{k+1}N_{k+1}$, where s_{k+1} ,f_{k+1}, N_{k+1}, and m_{k+1} are numbers that will be specified later. To show how to emulate f_k steps of a $s_k \times s_k$ mesh on a butterfly with N_k nodes, we first divide a $s_k \times s_k$ mesh into $s_{k+1} \times s_{k+1}$ slightly overlapping submeshes as shown in Figure 1. Then the butterfly is partitioned into subbutterflies of size N_{k+1}, and one submesh is assigned to a subbutterfly.

The emulation of the $s_k \times s_k$ mesh will be divided into f_k/f_{k+1} phases. In each phase we first attempt to run f_{k+1} steps of the emulation of each submesh in a subbutterfly. If nothing else were done, any node of a submesh at distance δ from the border of the submesh would not be able to be emulated for more than δ steps because the pebbles that it computes will depend on pebbles from another submesh. However, for every node v on the border of a submesh there is a node v' in another submesh emulating the same node of the mesh which will be able to successfully emulate f_{k+1} steps because it is located at distance $2f_{k+1}$ from the border of the submesh. We will show how to provide a path in the butterfly between the two nodes in the butterfly emulating v and v' of length $O(n_k)$. When the node emulating v' computes v's pebbles it will send copies of the pebbles to the node emulating v along this path. Once the node emulating v starts receiving pebbles from the node emulating v' it will resume the emulation. As the node emulating v resumes the emulation, nodes that were emulating nodes of the mesh that were waiting for pebbles from v will be able to resume their emulation. In order for all such pairs of nodes to be able to send pebbles back and forth simultaneously without slowing down the emulation, it will be necessary to choose the paths so that a most a constant number of paths will share an edge, and this must be true simultaneously for all levels of the recursion. In order to provide the paths connecting nodes in the butterfly, we will not use all subbutterflies in the partition of the butterfly for emulating submeshes; some subbutterflies will be used only for providing connections between subbutterflies.

We now describe how to embed the nodes of the mesh in the butterfly and to choose the paths connecting copies of nodes.

So now suppose that we have chosen the embedding of the nodes of a $s_{k+1} \times s_{k+1}$ mesh in a N_{k+1} node butterfly and the paths connecting corresponding nodes within the subbutterfly We will further require that for each node v on the border or at distance $2f_{k+1}$ from the border of the $s_{k+1} \times s_{k+1}$ mesh (we will refer to the set of all such nodes as F_{k+1}), that there is some node of the butterfly $< 0, x_v >$ and a path that connects $< 0, x_v >$ to a node $< i, y_v >$ that emulates v in the butterfly such that pebbles can be sent between $< 0, x_v >$ and $< i, y_v >$ without slowing down the simulation of the $s_{k+1} \times s_{k+1}$ mesh. Furthermore, x_v will have the property that $b_{\epsilon_{k+1}-1} \cdots b_0$ equals $10 \cdots 0$ where ϵ_{k+1} is a number that will be specified later, and for all v in F_{k+1}, their values of x_v will share a common value of $b_{2\epsilon_{k+1}-1} \cdots b_{\epsilon_{k+1}}$ that can be chosen arbitrarily, where $b_{n_{k+1}-1} \cdots b_0$ is the binary representation of x_v. We again divide a $s_k \times s_k$ mesh into submeshes as decribed in Figure 1. Now however, we modify this method for dividing the mesh into submeshes. We wish to require that all nodes in F_k in the mesh will lie in F_{k+1} for any submesh in which they are contained. In order to do this, we will shrink the sizes of submeshes in at most two rows and two columns of submeshes. When we recursively emulate f_{k+1} steps of a submesh that has had its size reduced we will consider it to be part of a larger mesh that has dummy nodes.

We will now partition the butterfly with N_k nodes into subbutterflies with N_{k+1} nodes. For a node in a butterfly we will denote the binary representation of the node's PIL by $b_{n_k-1}\cdots b_0$. Each subbutterfly will consist of all nodes of the butterfly with the following properties: there exists α such that α is a multiple of n_{k+1} possibly zero and such that all nodes in the subbutterfly share common values of $b_{\alpha-1}\cdots b_0$ and $b_{n_k}\cdots b_{\alpha+n_{k+1}}$, and $\alpha \leq i \leq \alpha + n_{k+1} - 1$.

Subbutterflies will be used to emulate submeshes. However, we will not use all butterflies to emulate submeshes; some subbutterflies will be used to create connections between subbutterflies that will be simulating submeshes. We will not use a subbutterfly if there there exists γ such that γ is a multiple of n_{k+1}, $\gamma > \alpha$ (where α is the α used to describe the nodes in the subbutterfly) and $b_{\gamma+\epsilon_{k+1}-1}\cdots b_\gamma$ equals the string $10\cdots0$ for all nodes in the subbutterfly, or if $\alpha > 0$ and $b_{\epsilon_{k+1}-1}\cdots b_0$ equals the string $10\cdots0$ or $0\cdots0$.

We must make sure that the number of subbutterflies to be used for simulating submeshes is greater than or equal to the number of submeshes to be emulated. The number of submeshes is at most

$$\left(\frac{s_k}{s_{k+1}-2f_{k+1}}+2\right)^2$$

(the additive two is due to the shrinkage of the size of some submeshes). The total number of subbutterflies in the partition of the butterfly is N_k/N_{k+1}. The number of subbutterflies that will not be used for simulating submeshes is at most

$$\left(\frac{n_k}{n_{k+1}}\right)^2 2^{n_k-n_{k+1}-\epsilon_{k+1}}.$$

Thus there will be enough subbutterflies if

$$\left(\frac{s_k}{s_{k+1}-2f_{k+1}}+2\right)^2 \leq \frac{N_k}{N_{k+1}} - \left(\frac{n_k}{n_{k+1}}\right)^2 2^{n_k-n_{k+1}-\epsilon_{k+1}}. \qquad (1)$$

We now describe how to choose the paths. We wish to choose a path connecting two nodes u and u' that are emulating nodes v and v'. We will again use α to describe the subbutterfly in which u is located as we did previously. Since v is in F_{k+1} for its submesh, we know that there exists some node u_1 in the subbutterfly such that u_1's level in the butterfly is α (it has level zero when considered as part of the subbutterfly), and such that the bits $b_{\alpha+\epsilon_{k+1}-1}\cdots b_\alpha$ in u_1's PIL equals $10\cdots0$, and such that for all v in F_{k+1} in the submesh, the PIL's for the respective PIL's of u_1 share a common value of $b_{\alpha+2\epsilon_{k+1}-1}\cdots b_{\alpha+\epsilon_{k+1}}$ that can be chosen arbitrarily. We choose $b_{\alpha+2\epsilon_{k+1}-1}\cdots b_{\alpha+\epsilon_{k+1}}$ to be $b_{\epsilon_{k+1}-1}\cdots b_0$, which is the same for all nodes in u's subbutterfly. We now choose u_2 to be the node in the butterfly at level zero whose PIL is obtained by converting the ϵ_{k+1} least significant bits of u_1's PIL to $10\cdots0$. By our choice of subbutterflies to be used for simulating submeshes and our choice of $b_{\alpha+2\epsilon_{k+1}-1}\cdots b_{\alpha+\epsilon_{k+1}}$ for u_1's PIL we know that the paths from u_1 to u_2 for different choices of v are disjoint. One similarly chooses nodes in the buttefly u_1' and u_2' for v'. For all choices of u_2 and u_2' we now choose paths connecting u_2 and u_2' by routing a permutation through nodes of the butterfly which have PIL's whose ϵ_{k+1} least significant bits equals $10\cdots0$ using one pass up through the butterfly and one pass down [1]. None of these paths will conflict with any previously chosen paths.

To finish the description of the embedding, we must show that for each node v in F_k in the mesh being emulated, that there is a node u in the butterfly that can be connected by a path of length $O(n_k)$ to some node w in the butterfly which is emulating v so that pebbles can be sent from u to w or w to u without slowing down the simulation of the mesh, and such that u is chosen so that it has level zero, that the ϵ_k least significant bits of its PIL equal $10\cdots0$, and so that $b_{2\epsilon_k-1}\cdots b_{\epsilon_k}$ is some arbitrarily chosen number that is common for all u. We first assign nodes in the mesh in F_k to nodes in the butterfly with the required characteristics, so that at most one node of the mesh is assigned to a node in the butterfly. For this to be possible there must be enough nodes in the butterfly with the required properties, and this will be true if

$$\log 8s_k < n_k - 2\epsilon_k. \qquad (2)$$

We already know that for a node v in F_k, that there will be some node u' in the butterfly with level zero whose ϵ_{k+1} least significant bits equal $10\cdots0$ which is connected by a path of length $O(n_k)$ to w; this is because when we divided the mesh into submeshes, we required v to be located in F_{k+1} of any submesh in which it was contained, and we have previously described a path from w to the desired node u'. We now again connect all corresponding pairs of u's and u''s using permutation routing as before.

We now choose the values of s_k, f_k, N_k, ϵ_k and m_k so that (1) and (2) are satisfied. We first denote by $\omega(N)$ the smallest value of k such that $N^{20^{-k}} < 2$. We let $\epsilon_k = \frac{1}{30}\log n_k$. We let $s_0 = \sqrt{N}$, and for $k > 0$, choose s_k and n_k so that $N^{10^{-k}} \leq s_k \leq (N^{10^{-k}})^2$, n_k is a power of two ($N_k = n_k 2^{n_k}$), and

$$s_k^2 = N_k \prod_{j=k+1}^{\omega(N)} m_j,$$

where

$$m_k = \left(\frac{s_k}{s_k - 2f_k} + 2\right)^2 \left(\frac{1}{1 - \frac{n_k}{n_{k+1}} 2^{-t_{k+1}}}\right)$$

We know that we can choose such a s_k since for all possibles values of s_k in the specified range the product

$$\prod_{j=k+1}^{\omega(N)} m_j.$$

is bounded. We also choose $f_0 = T$, $f_1 = \min\{T, \sqrt{s_1}\}$, and for $k \geq 2$, $f_k = \sqrt{s_k}$.

We now consider the time required for the emulation. Let T_k be the time to emulate f_k steps of a $s_k \times s_k$ mesh on a N_k node butterfly. The emulation is divided into f_k/f_{k+1} phases. Each phase requires time $T_{k+1} + O(n_k)$ and n_k is $O(\log s_k)$. Thus

$$T_k = \frac{f_k}{f_{k+1}}(T_{k+1} + O(\log s_k))$$

and therefore the total time for the emulation

$$\begin{aligned} & \sum_{k=0}^{\omega(N)} \left(\prod_{j=0}^{k} \frac{f_k}{f_{k+1}}\right) \log s_k \\ = & \sum_{k=0}^{\omega(N)} \frac{f_0 \log s_k}{f_{k+1}} \\ = & \ T \sum_{k=0}^{\omega(N)} \frac{\log s_k}{f_{k+1}} \\ = & \ O(T). \end{aligned}$$

□

5.3 A constant congestion embedding of the shuffle-exchange graph in a butterfly

In this section, we show how to embed an N-node shuffle-exchange graph in an $O(N)$-node butterfly graph with constant congestion and $O(\log N)$ dilation.

The N-node shuffle-exchange graph is defined for every N which is a power of two. Each node of the ($N = 2^k$)-node shuffle-exchange graph is associated with a unique k-bit binary string $a_{k-1}...a_0$. We call this string the *label* of the node. Two nodes, w and w', are linked via a *shuffle* edge if w' is a left or right cyclic shift of w. Two nodes, w and w', are linked via an *exchange* edge if w and w' differ in the least significant bit, a_0.

A constant congestion embedding requires that very few edges of the shuffle-exchange be mapped to long (more than constant length) paths in the butterfly. In addition, these paths must not overlap each other very often. To ensure this, we use the afore-mentioned theorem of Benes concerning a butterfly graph's ability to embed a permutation on its inputs.

That is, if the set of long paths can be decomposed into a constant number of (partial) permutations of the inputs of the butterfly, the long paths can be embedded with constant congestion. It is easy to see that we can embed the long paths in this manner when there are at most a constant number of endpoints of long paths in any single butterfly row. (We route a path from each endpoint to to the input of its row. This leaves us with a constant number of "Benes routings" to perform.)

So we map the nodes of a shuffle-exchange graph to the nodes of a butterfly graph so that

1. at most a constant number of shuffle-exchange nodes are mapped to any one butterfly node, and

2. each butterfly row contains at most a constant number of shuffle-exchange nodes which have any neighbor mapped to a distant node in the butterfly.

Short paths only contribute constant congestion since they have constant length. Long paths only contribute constant congestion since we can route any permutation with congestion 2, and we only need to route a constant number of (partial) permutations. Also, the length of the short paths is constant and the long paths is $O(\log n)$.

In particular, we map the nodes of a $N = 2^n$-node shuffle-exchange graph to the nodes of a $(n + 2 - \log n)2^{n+2-\log n} \approx 4N$-node butterfly graph. Each node in this N-node shuffle-exchange graph has n bits in its label. A node in the butterfly can be specified by a row represented by $n + 2 - \log n$ bits, and a level in the row. The level in the row corresponds to a bit that can be flipped to enter another row. Thus, we first associate a shuffle-exchange node with a particular row of the butterfly by removing $\log n - 1$ adjacent bits of its label none of which are the least significant bit, then we pick the level in the row which corresponds to where the least significant bit of the shuffle-exchange node appears in the row's representation.

We map a shuffle-exchange node w to a node in the butterfly as follows,

1. Consider the longest string of zeros in w ignoring the least significant bit, break ties by choosing the first one from the left.

2. Pick out $\log n - 1$ bits as follows;

   (a) If possible choose the $\log n - 1$ bits after the zeros and before the lsb,

   (b) otherwise if possible choose the $\log n - 1$ bits preceding the longest string of zeros,

   (c) otherwise choose the last $\log n - 1$ bits of the string of zeros (note that in this case more than $n - 2\log n$ bits are zeros).

3. Treat these bits as a number (it will be in the range $0 \ldots \frac{n}{2}$), call this number s, and the sequence of bits a_s.

4. Remove the bits of s from w, extend the chosen string of zeros on the right (left) by a 01 (10) if the bits were removed from the right (left) of the block of zeros, and cyclic shift the resulting string so that s bits appear after the longest string of zeros, this specifies the row.

Symbolically, we map $w = z0^k a_s yb$ to row $u0^{k+1}1v$, or we map $w = za_s 0^k yb$ to row $u10^{k+1}v$, with $ybz = vu$ and $|v| = s$. (Note that we map to a row with a unique longest string of zeros not straddling the bit which is at the level of the butterfly node.) It is easy to see that the least significant bit of w, b, is somewhere in the representation of the row. We choose the level in the row to correspond to the position of b in the row's representation.

We must argue that the mapping achieves condition 1 and 2 above.

First, we introduce some more notation. We define a *necklace* to be a set of shuffle-exchange nodes which are connected only by shuffle edges. Alternatively, a necklace is a set of nodes having labels which are cyclic shifts of each other. A *necklace's label* is the lexicographically minimum label of its nodes. We can specify a shuffle-exchange node by the label of its necklace and the position of the least significant bit of the node's label in the necklace's label.

We define the *domain of a butterfly node* to be the set of shuffle-exchange nodes that are mapped to it by our mapping.

Now we show that the mapping is at most two to one. That is, given a butterfly node $\langle p, r\rangle$ we can describe at most two shuffle-exchange nodes that could possibly be mapped to $\langle p, r\rangle$ as follows. Recall that a butterfly node $\langle p, r\rangle$ has all the bits of w in r's binary representation except for a_s. And these, we recover by finding the length of the string after the longest group of zeros in r's binary representation not straddling the pth bit. We know that we have to reinsert them either directly before or directly after that group of zeros. This gives us all the bits of the domain nodes except for a cyclic shift uncertainty. Thus, the domain of $\langle p, r\rangle$ can only be nodes from two necklaces. Furthermore, the least significant bit of the nodes' labels is uniquely specified by the place where the pth bit of r's binary representation occurs in the necklaces' labels. Thus only two shuffle-exchange nodes can be mapped to any node in the butterfly.

Finally, we argue that we map at most a constant number of shuffle exchange nodes with distant neighbors to any butterfly row.

Notice that we always ignore the value of the least significant bit in the mapping of shuffle-exchange nodes to butterfly nodes. Thus the mapping maps two shuffle-exchange nodes to two nodes that only differ in the bit that can currently be changed by a butterfly edge. Thus, any exchange edge needs only flip the bit at the node's level, which only requires a path of length 2. Thus all exchange edges are embedded in short paths.

Now consider the shuffle edges. We show that at most a constant number of shuffle edges leave any row of the butterfly. (It is easy to see that all the shuffle edges in a row are mapped to single edges in the butterfly graph.) Again, consider the inverse mapping of a butterfly node, $\langle p, r\rangle$, to two shuffle-exchange nodes. The necklaces of the domain nodes of row r's nodes, are the same for most of the row. They change only at certain transition levels in the row; levels, p, in the row where the position of the longest string of zeros not straddling p changes, or levels in the row where we become unsure or sure of which side of the zeros to replace the removed bits, a_s.

The position of the longest string of zeros not straddling p only changes at two points; inside the row's unique longest string of zeros. When the row level is within $\log n$ bit positions to the right of the longest string of zeros, we know that pieces of two shuffle-exchange necklaces could have been mapped to the row. Outside this range we know that only one necklace is mapped to the row: Inside the group of zeros the bits were definitely taken out before the group of zeros, and further to the right they were definitely taken out after the group of zeros. Thus entering this stretch and leaving this stretch gives us two more bad levels. Thus we have four transition levels in all, and for each of these at most four necklaces could enter or leave the row at any of these levels. Thus at most 16 long shuffle edges can have endpoints in this row. (Careful counting can reduce this number to 6.)

Thus at most 16 long edges are adjacent to any row of the butterfly. This satisfies condition 2, above.

Thus, the shuffle-exchange graph can be embedded in the butterfly with constant congestion.

5.4 Application to optimal area and volume layouts for the shuffle-exchange graph

The N-node butterfly can be laid out in $O(N^2/\log^2 N)$ area (trivially) and in $O(N^{3/2}/\log^{3/2} N)$ volume [20]. Since the N-node shuffle-exchange graph can be embedded in the N-node butterfly with constant congestion, we can simply blowup these layouts by a constant factor to obtain layouts for the shuffle-exchange graph with equivalent area and volume.

5.5 A work preserving emulation of a shuffle-exchange graph

We construct an $O(\log N)$- step work-preserving simulation of the shuffle-exchange graph on the butterfly by first embedding the shuffle-exchange graph in an $N \log N$-node butterfly with constant congestion, and then embedding the $N \log N$-node butterfly in an N-node butterfly in the natural way. It is not difficult to show that the N-node butterfly can then simulate the $N \lg N$-node shuffle-exchange in $O(\log N)$ steps. Whether or not there is a real-time emulation remains an interesting open question.

6 Emulations in a shuffle-exchange graph

6.1 Work preserving emulations of arbitrary binary trees

It is well known that the shuffle-exchange graph can emulate a complete binary tree in real time. Thus by the results of Section 4, we know that there is an $O(\log \log N)$-time work-preserving emulation of the class of binary trees on the shuffle-exchange graph. Whether or not this emulation can be made real-time remains an open question.

6.2 A constant-dilation embedding of N^ϵ distinct $N^{1-\epsilon}$-node butterflies

A shuffle-exchange graph of size N can hold N^ϵ distinct $N^{1-\epsilon}$-node butterfly graphs for $0 < \epsilon < 1$ with max load and congestion of $O(1/\epsilon)$.

We illustrate this by proving it for $\epsilon = 1/2 - \log(1/2 \log N)$. That is, we embed $M/\log M$ distinct $M \log M$-node butterfly graphs in an $N = M^2$-node shuffle-exchange graph with constant congestion and constant dilation. We assume that $M = 2^k$. Thus each row of the butterfly can be represented by a k-bit string, and each node of the shuffle-exchange can be represented by a $2k$-bit string. A similar result was proved by Raghunathan and Saran [16].

To map $M/\log M$ butterflies to the shuffle-exchange graph, we use the following easily proven lemma.

Lemma 16 *The set of $k = \log M$-bit strings has at least $M/2 \log M$ nonintersecting subsets of $\log M$ distinct strings which are cyclic shifts of each other.*

For each of these groups we pick the lexicographically minimum string to represent the group. We associate the $M/\log M$ butterflies two to one with the $M/2\log M$ groups' representative strings. Say butterfly i is associated with string w^i. We map a node (p, r) in butterfly i to a shuffle-exchange node by shuffling the bits of w_i with the bits of r's representation, and choosing the current bit to be under the image of r_p. That is, node (p, r) in butterfly i is mapped to shuffle-exchange node $r_1 w_1^i \ldots \underline{r_p} w_p^i \ldots r_n w_p^i$.

From a shuffle-exchange node we can recover the representative string w_i by picking out every other bit and shifting to the lexicographically minimum string. We finding the row string by picking out the other bits and shifting by the same amount. The position in the row is clearly the number of shifts we used to get to w_i and the row number.

To finish, we observe that each edge in any of the butterflies is mapped to a path of length at most three in the shuffle-exchange graph since we either shift twice to reach $(p+1, r)$'s image, or we exchange the current bit and shift twice to reach $(p+1, r_1 .. \bar{r}_p ... r_n)$'s image.

Thus we can embed $\sqrt{N}/\log\sqrt{N}$ $\sqrt{N}\log\sqrt{N}$-node butterflies in an N-node shuffle-exchange with max load 2, and dilation 3.

This technique can be extended to prove that for any constant $0 < \epsilon < 1$, N^ϵ distinct $N^{1-\epsilon}$ butterfly graphs can be embedded in an N-node shuffle-exchange.

6.3 Application to sorting on a shuffle-exchange graph

It is known that an N-node butterfly can sort N packets with high probability in $O(\log N)$ steps [11, 15, 17]. The result does not directly extend to the shuffle-exchange graph because the shuffle-exchange graph does not have the nice recursive structure possessed by the butterfly. However, by combining the embedding result of Section 6.2, the butterfly sorting algorithm in [11], and the columnsort algorithm of [9], we can obtain an algorithm for sorting N packets on an N-node shuffle-exchange in $O(\log N)$ steps with high probability.

6.4 Real time emulations of arrays

By combining a single level of the kind of analysis in Section 5.2 with the result of Section 6.2, we can emulate an array in real time on a shuffle-exchange graph. This is despite the fact that any $O(1)$ to 1 embedding of an N-node array (with dimension 2 or more) in a shuffle exchange graph has dilation $\Omega(\log \log N)$ [2].

6.5 A work preserving emulation of the butterfly

By using standard techniques in routing normal hypercube algorithms, it is easily shown that there is an $O(\log N)$-step work-preserving simulation of a butterfly on a shuffle-exchange graph. Whether or not there is a real-time simulation remains an important open question.

7 Remarks and open questions

There are many questions left open by this paper. We list a few of them in what follows.

1. Is there a real-time simulation of a complete ternary tree on a complete binary tree?

2. Is there a (universal) class of bounded-degree graphs that can simulate the class of all bounded-degree graphs? (If so, they must be expanders.)

3. Is there a real-time simulation of a butterfly on a shuffle-exchange graph or vice-versa?

4. Can the notion of work-preserving be meaningfully modified to incorporate measures such as VLSI layout area?

5. Are meaningful results possible if we consider simulations that are not work-preserving, but which are close to work-preserving (e.g., we allow inefficiency of $\Theta(\log N)$)?

Acknowledgements

We are deeply indebted to Marc Snir for his helpful comments and for motivating this research. Thanks also to Tom Cormen for producing Figure 1.

References

[1] V. E. Benes, "Optimal rearrangeable multistage connecting networks," *Bell System Technical Journal*, Vol. 43, July 1964, pp. 1641-1656.

[2] S. N. Bhatt, F. R. K. Chung, J.-W. Hong, F. T. Leighton, and A. L. Rosenberg, "Optimal simulations by butterfly networks," *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, May 1988, pp. 192–204.

[3] S. N. Bhatt, F. R. K. Chung, F. T. Leighton, and A. L. Rosenberg, "Optimal simulations of tree machines." *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, IEEE, October 1986, pp. 274–282.

[4] S. N. Bhatt and I. Ipsen, *Embedding Trees in the Hypercube*, Yale Univ. Report RR–443.

[5] D. S. Greenberg, L. S. Heath, and A. L. Rosenberg, "Optimal embeddings of the FFT graph in the hypercube," unpublished manuscript.

[6] D. Hoey and C. E. Leiserson, "A layout for the shuffle-exchange network," *Proceedings of the 1980 International Conference on Parallel Processing*, IEEE, August 1980, pp. 329–336.

[7] D. J. Kleitman, F. T. Leighton, M. Lepley, and G.L. Miller, "New layouts for the shuffle exchange graph," *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, May 1981, pp. 278–292.

[8] C. P. Kruskal, L. Rudolph, and M. Snir, "A complexity theory of efficient parallel algorithms," unpublished manuscript.

[9] F. T. Leighton, "Tight bounds on the complexity of parallel sorting," *IEEE Transactions on Computers*, Vol. C-34, No. 4, April 1985, pp. 344–354.

[10] F. T. Leighton, M. Lepley, and G. L. Miller, "Layouts for the shuffle-exchange graph based on the complex plane diagram," *SIAM Journal of Algebraic and Discrete Methods*, Vol. 5, pp. 177–181.

[11] T. Leighton, B. Maggs, and S. Rao, "Universal packet routing algorithms," *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, IEEE, October 1988, pp. 256–271.

[12] F. T. Leighton and G. L. Miller, "Optimal layouts for small shuffle-exchange graphs," *VLSI 81-Very Large Scale Integration*, ed. J. Gray, Academic Press, London, 1981, pp. 289–299.

[13] F. Meyer auf der Heide, "Efficient simulations among several models of parallel computers," *SIAM Journal on Computing*, Vol. 15, No. 1, February 1986, pp. 106–119.

[14] C. H. Papadimitriou and M. Yannakakis, "Towards an architecture-independent analysis of parallel algorithms," *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, May 1988, pp. 510–513.

[15] N. Pippenger, "Parallel communication with limited buffers," *Proceedings of the 25th Annual Symposium on Foundations of Computer Science*, IEEE, October 1984, pp. 127–136.

[16] A. Raghunathan and H. Saran, "Is the shuffle-exchange better than the butterfly?," unpublished manuscript.

[17] J. H. Reif and L. G. Valiant, "A logarithmic time sort for linear size networks." *Journal of the Association for Computing Machinery*, Vol. 34, No. 1, January 1987, pp. 60–76.

[18] M. Sekanina, "On an ordering of the set of vertices of a connected graph," *Pub. Faculty of Sci. Univ. Brno*, Czechoslovakia, No. 412, 1960, pp. 137–142.

[19] D. Steinberg and M. Rodeh, "A layout for the shuffle-exchange network with $\Theta(N^2/\log^{3/2} N)$ area", *IEEE Transactions on Computers*, Vol. C-30, No. 12, December 1981, pp. 977–982.

[20] D. S. Wise, "Compact layouts of banyan/FFT networks," *VLSI Systems and Computations*, H. T. Kung, B. Sproull and G. Steele, eds., 1981, pp. 186–195.